Non-generic floating-point software support for embedded media processing

Claude-Pierre Jeannerod
INRIA
Laboratoire LIP (CNRS, ENSL, INRIA, UCBL),
Université de Lyon, France
claude-pierre.jeannerod@ens-lyon.fr

Jingyan Jourdan-Lu*, Christophe Monat
STMicroelectronics
Compilation Expertise Center, Grenoble, France
jingyan.jourdan-lu@st.com, christophe.monat@st.com
* also a member of Laboratoire LIP

***Abstract*—This paper presents some work in progress on the design and implementation of efficient floating-point software support for embedded integer processors. We provide quantitative evidence of the benefits of supporting various non-generic (that is, specialized, fused, or simultaneous) operations in addition to the five basic arithmetic operations: for individual calls, speedups range from 1.12 to 4.86, while on DSP kernels and benchmarks, our approach allows us to be up to 1.34x faster.**

***Index Terms*—embedded integer processor; floating-point arithmetic; fused floating-point operations; VLIW architecture; instruction level parallelism; C software implementation.**

I. Introduction

Even though media processing applications may rely intensively on floating-point computations, some modern embedded media processors such as the ST231 from the STMicroelectronics ST200 VLIW family do not contain floating-point hardware and provide architectural support only for integer arithmetic. This choice avoids paying high costs on silicon surface and power consumption.

Yet, this trade-off has to be compensated: a first approach would be to convert the applications to some fixed-point [1] or block floating-point format [2]. With such techniques, ensuring accuracy can however be fairly complex, and eventually costly and unsustainable. A second approach, which is the one we favor here, consists in designing a high-performance floating-point support.

The design and implementation of such a software library is critical in several aspects: not only its performance must enable key applications to reach an acceptable performance level, but the compliance with the IEEE 754-2008 standard [3] must not be compromised.

In order to achieve good-enough performance without sacrificing for accuracy, a first step is to optimize the five basic arithmetic operations by taking into account some features of the target architecture (parallelism, large multipliers, leading-zero counters,...). This was achieved by FLIP 1.0 (Floating-point Library for Integer Processors) [4], with new algorithms exposing high instruction-level parallelism (ILP). That library thus better exploits the VLIW architecture of the ST231 processor than the reference library SoftFloat [5] and in practice, the latency of each operator on ST231 was reduced by a factor of 1.85 to 5.21, as the following table shows:

	$+$	$-$	$\times$	$/$	$\sqrt{}$
SoftFloat	48	49	31	177	95
FLIP 1.0	26	26	21	34	23
speedup	1.85	1.88	1.48	5.21	4.13

Such speedups (which are given here assuming single precision, rounding 'to nearest even', and subnormal support) have been achieved by a combination of techniques, among which an optimized use of the IEEE 754 format encodings, a novel algorithmic approach based on highly-parallel polynomial evaluation for computing accurate approximations of functions like division and square root, and also some compiler optimizations; interestingly enough and unlike what is sometimes believed, the overhead for supporting subnormals (that is, the tiny floating-point numbers allowing gradual underflow) turned out to be extremely reasonable (for example, 5 cycles out of 34 for division and 2 out of 23 for square root) [6], [7].

However, embedded processing application codes and benchmarks typically involve numerical blocks that exhibit particular patterns, which we may refer to as being *non-generic*. For example, Euclidean norm calculations consist of square rooting a sum of squares, and radix-2 FFTs, arithmetic over the complex numbers, and geometric predicates use dot products in dimension two (DP2). Thus, a second step to increase further the performances of such applications on integer processors like the ST231 is to design optimized non-generic operators like square and DP2, that will then be added to the existing basic arithmetic software support and selected at compile time. We can group non-generic operators into three categories, defined as follows:

- A *fused operator* replaces a set of two or more floating-point operators by a single one. Examples include the fused multiply-add (FMA) operation $xy + z$, as well as DP2 mentioned above.
- A *specialized operator* replaces a generic operator when the compiler can prove properties about its arguments. A typical example is that of square replacing a generic product xy whenever x equals y.
- A *pair of operators* simultaneously evaluates two operators on the same input. For example, given floating-point input x and y, the so-called addsub operator will compute the pair $(x + y, x - y)$.

Note that some fused operators may in fact be fully specified by a standard like IEEE 754-2008. For example, this is the case of FMA and of functions like reciprocal square root and hypotenuse which compute, respectively, $1/\sqrt{x}$ and $\sqrt{x^2+y^2}$ with just one rounding error; see [3, Table 9.1].

In hardware, some non-generic operators have been studied extensively. For example, we refer to [8], [9], [10], [11] and the references therein for DP2 and addsub designs applied to FFT butterfly units. (Note that in these works addsub is called 'fused' rather than 'pair'.) Another example is the computation of qth roots, for which algorithms and architectures have been proposed in [12]. In addition, customization is now common practice for FPGAs due to the high flexibility offered by such architectures [13].

In software, several non-generic operators have been considered for processors having floating-point hardware capabilities [14], [15]. For integer processors like the ST231, division has been specialized to reciprocal [6, p. 4] and examples of fused operators include reciprocal square root [16] as well as third and fourth roots [6]. Such operators are however not always critical ones in DSP applications, where most of the computation time is spent on FMA-like patterns, that is, expressions made of additions and various kinds of multiplications (generic, squares, by small constants).

Thus, in this paper we study a set of 11 non-generic operators, described in Section II and including in particular the aforementioned FMA, DP2, square, and addsub operators. We provide quantitative evidence of the benefits of their optimized implementation by analyzing in Section III the performance gains achieved both via individual calls and on the UTDSP benchmark suite. Although we focus here on single precision (which is the floating-point format used for benchmarks like UTDSP), we comment in Section IV on the possibility of going beyond this on ST231 by implementing non-generic operators in double or quadruple precision.

II. Operators currently implemented

So far, 4 fused operators, 5 specializations, and 2 operator pairs have been designed and implemented (in standard ANSI C11), for IEEE 754 single precision (called binary32 format in [3]) and with full support of subnormal numbers, signed zeros, signed infinities, and NaNs (Not-a-Numbers). We provide correct rounding for all the operators listed below except for sine and cosine, which for cost reasons are implemented with a guaranteed accuracy of only 1 unit in the last place. Furthermore, correctly-rounded results can be obtained for any of the four rounding modes required by the standard: to nearest even (RN), up (RU), down (RD), and to zero (RZ).

A. Fused operators

We have implemented the following four fused operators. Each of them commits just one rounding error.

- **FMA** (fused multiply-add): $xy+z$.
- **FSA** (fused square-add): x^2+z with $z \geqslant 0$.
- **DP2** (dot product in dimension two): $xy+zt$.
- **SOS** (sum of two squares): x^2+y^2.

The FMA belongs to the IEEE 754 standard since its 2008 revision [3, §5.4.1] and is a key operation for linear algebra and schemes like Horner's rule and Newton's method. However, due its lack of symmetry, this operator is well-known to introduce subtle programming issues when evaluating expressions like DP2; see [17] and [18, §2.6]. This is why we also provide a correctly-rounded DP2 operator. We also provide FSA and SOS which appear in n-dimensional and 2-dimensional Euclidean norm calculations; in those cases the algorithms for FMA and DP2 can be simplified significantly, thus providing opportunities for acceleration.

B. Specialized operators

We have also implemented the following five special cases of multiplication and addition:

- **mul2** (multiplication by two): $2x$.
- **div2** (multiplication by one half): $\frac{1}{2}x$.
- **scalb** (multiplication by an integer power of two): $2^n x$ with n a 32-bit signed integer.
- **square** (squaring): x^2.
- **addnn** (addition of non-negative terms): $x+y$ with $x \geqslant 0$ and $y \geqslant 0$.

All these patterns appear in application codes and can be implemented much faster than generic multiplication or generic addition by means of specific algorithms. They are also fully specified by the IEEE 754 standard in the sense that mul2, div2, square, and addnn inherit the specification of multiplication and addition, and also since scalb is itself specified in [3, §5.3.3]. Furthermore, since we assume radix 2 floating-point arithmetic no rounding occurs for mul2 and scalb with $n \geqslant 0$. Finally, FSA and SOS can of course be also considered as a specialized versions of the fused operators FMA and DP2.

C. Pairs of operators

Two pairs of operators have been implemented so far, which typically occur in DSP kernels (FFT butterflies and rotations):

- **addsub** (simultaneous addition and subtraction): $(x+y, x-y)$.
- **sincos** (simultaneous sine and cosine over a reduced range): $(\sin x, \cos x)$ with $x \in [-\frac{\pi}{4}, \frac{\pi}{4}]$.

Such blocks give the opportunity to share some computations and also to expose more ILP in an easy way, thus allowing reduced latencies than by calling the two operations in sequence. For now, sincos assumes a reduced range for the input (see [19] for a detailed algorithm and error analysis). A range reduction step, which is well known to be common to both sine and cosine [15], is currently being developed and in this case the benefit of having an operator pair is even more obvious.

III. Experimental results obtained on ST231

A specific variant of the production compiler has been developed to support our experiments. The selection of the *fused operators* and the most simple variants of the *specialized operators* can be achieved at high level, on mostly syntactic criteria, but as there are many choices to pattern-match expressions, some heuristics are involved. The selection of the *paired*

operators is more elaborate since the expressions candidate for such an association may not belong to the same statements. Finally, the selection of the most elaborate form of *specialized operators* requiring specific conditions to be met, such as the positivity of the arguments for the FSA, require an elaborate static analysis phase to prove that the condition holds.

The measures of the operator performance are done with a cycle-accurate simulator, configured not to account for the I- or D- cache cycles: we call these 'perfect cycles'. For these kind of operators, the D-cache cycles are zero by design, and the I-cache cycles are rapidly amortized for these relatively small functions in floating-point intensive code.

A. Operator performances

Table I gives for each rounding mode and the first 10 non-generic operators the latency (in numbers of cycles) and code size (in numbers of integer instructions, and displayed within square brackets) obtained for one call on ST231.

	RN		RU		RD		RZ	
mul2	5	[11]	7	[15]	7	[15]	6	[13]
div2	7	[18]	7	[19]	7	[19]	6	[15]
scalb	15	[50]	16	[52]	16	[52]	12	[40]
square	12	[42]	11	[37]	9	[31]	9	[31]
addnn	15	[47]	15	[43]	14	[35]	14	[35]
FSA	22	[73]	22	[70]	19	[54]	19	[54]
FMA	42	[161]	42	[158]	42	[155]	39	[149]
SOS	26	[81]	25	[77]	22	[62]	22	[62]
DP2	51	[193]	50	[189]	50	[188]	47	[180]
addsub	28	[96]	30	[106]	30	[106]	26	[86]

TABLE I
LATENCIES IN # CYCLES [AND CODE SIZES IN # INTEGER INSTRUCTIONS].

For sine and cosine over the reduced range $[-\pi/4, \pi/4]$ the following latencies and code sizes have been achieved:

sine	cosine	sincos
19 [31]	18 [25]	19 [46]

A good measure of the exploitation of ILP is the Instruction Per Cycle ratio (IPC), which ideally should approach 4 for the 4-way ST200 VLIW core. We observe in Table I that in practice this range varies between a minimum of 2.14 for the mul2 operator in RU or RD, to 3.83 for FMA in RN and DP2 in RZ.

Then, a measure of the efficiency of the non-generic operators compared to their naive implementation[1] can be summarized as speedups ('performance of the original code' / 'performance of the improved code'), and Code Reduction Ratio (CRR), defined as 'size of the improved code' / 'size of the original code.' In this way, speedups > 1 denote an acceleration, while CRRs < 1 indicate a code size reduction. Table II displays these ratios for the RN mode (except for sincos), but the results are essentially the same for the three other rounding modes. The speedups range from 1.12 to 4.86, while CRRs can be as low as 0.15. It is worth noting that the FMA's adverse CRR is due to bigger alignment logic in the addition stage, which is necessary for correct rounding.

[1]For example, the naive implementation of $xy + z$ consists of one generic multiplication followed by one generic addition; a naive implementation of sincos consists in one call to sine followed by one call to cosine.

	Speedup	CRR
mul2	4.2	0.15
div2	4.86	0.17
scalb	1.4	0.70
square	1.75	0.49
addnn	1.73	0.54
FSA	2.14	0.46
FMA	1.12	1.02
SOS	2.62	0.35
DP2	1.33	0.84
addsub	1.86	0.56
sincos	1.95	0.82

TABLE II
SPEEDUPS AND CODE REDUCTION RATIOS FOR RN.

To conclude this first set of experiments, Table III illustrates with the example of multiplication that naive specialization is not enough to obtain our results, and that entirely different algorithms have been needed. Here 'naive specialization' means that 2 or x has been substituted to y in the C code of generic multiplication xy with, say, RN. After compilation of these new codes, we see a latency reduction of only 2 cycles.

Operation	Generic	Naive specialization	Optimized specialization
$2x$	21	18	5
$\frac{1}{2}x$	21	19	7
x^2	21	19	12

TABLE III
NAIVE VS OPTIMIZED SPECIALIZATION (IN # CYCLES AND FOR RN).

B. Performances on the UTDSP benchmark

The UTDSP Benchmark Suite [20] was created by Corinna G. Lee at University of Toronto, to assess C compilers efficiency on typical DSP code. It is divided in two classes: *kernels* (FFTs, filters,...) and *applications* (LPC coding,...). The code is provided in multiple styles (usage of arrays versus pointers), but this is now irrelevant to modern compilers that produce the same performance level for all styles. It is quite representative of our application domain and has been found, notably on compiler optimization work, to be a good predictor of improvements that can be obtained at a larger scale, on actual applications.

Table IV summarizes the gains on various UTDSP kernels and applications, wich are:

- **FFT-256** (resp. **FFT-1024**), a complex radix-2 decimation-in-time 256-point (resp. 1024-point) FFT;
- **Latnmr-8** (resp. **Latnmr-32**), an 8th order (resp. 32nd) normalized lattice filter processing 1 (resp. 8) point(s);
- **SPE**, a power spectral estimator using the periodogram averaging method;
- **ADPCM**, an Adaptative Differential PCM encoder;
- **LPC**, a Linear Predictive Coding encoder.

Table IV has been built by using internal compiler options that enable selection of FMA operators only, and options that enable the selection of the full set of non-generic operators.

	FLIP 1.0	FMA		non-generic operators	
FFT-256	317857	298401	[1.07]	266657	[1.19]
FFT-1024	1579207	1481927	[1.07]	1323207	[1.19]
Latnrm-8	1842	1644	[1.12]	1388	[1.33]
Latnrm-32	467365	409189	[1.14]	347685	[1.34]
SPE	1189958	1101236	[1.08]	966556	[1.23]
ADPCM	1733749	1618927	[1.07]	1566759	[1.11]
LPC	984989	880762	[1.12]	880403	[1.12]

TABLE IV
NON-GENERIC OPERATORS VS FLIP 1.0 IN # CYCLES AND [SPEEDUPS].

The selection of the FMA alone brings a speedup of up to 1.14, as expected, on par with the FMA speedup itself (given in Table II). Then, enabling the full set of fused operators brings in most cases a speedup that can be as high as 1.34. The usage of the non-generic operators that replace FMA or complement it is displayed in Table V.

	non-generic operators selected
FFT-256,1024	DP2, FMA
Latnrm-8,32	DP2, FMA
SPE	DP2, FMA, SOS, addsub
ADPCM	DP2
LPC	FMA, mul2, square, addsub

TABLE V
NON-GENERIC OPERATORS SELECTED.

For these benchmarks, the rounding mode is not changed and defaults to RN, but the performance would obviously be even better by using RZ.

IV. CONCLUDING REMARKS

In this paper we have shown the benefits of having optimized software implementations of a set of 11 non-generic operators, to significantly speed up both individual calls and application codes.

Although we have focused on single precision, our approach scales to higher precisions like double or quad (called binary64 and binary128 floating-point formats in [3]) as soon as we have software support for 64-bit or 128-bit integer arithmetic. In fact, to facilitate engineering, the design of all our operators has been parametrized by the format and generically described in an XML-based scheme, that is used to generate the C source code of all variants for all formats and rounding modes.

For now the 64-bit integer layer exists naturally in the compiler which supports the ANSI C11 'long long' type, and for which lots of efforts have been done for code generation and optimization. One of the reasons is that this layer is intensively used for high precision fixed-point computations when the floating-point support and optimization fall short.

The 128-bit support exists only as a prototype library with no native implementation in the compiler: thus at the moment it is not as efficient as it could be, suffering from the representation of 128-bit types by structures. Ideally, implementing the native support for the __uint128_t and __int128_t gcc extensions (at the moment limited to 64-bit machines) would alleviate these issues.

REFERENCES

[1] D. Ménard, D. Chillet, F. Charot, and O. Sentieys, "Automatic floating-point to fixed-point conversion for DSP code generation," in *Proceedings of the 2002 international conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2002, pp. 270–276.

[2] A. Mitra, M. Chakraborty, and H. Sakai, "A block floating-point treatment to the LMS algorithm: efficient realization and a roundoff error analysis," *IEEE Transactions on Signal Processing*, vol. 53, no. 12, pp. 4536–4544, 2005.

[3] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008.

[4] C.-P. Jeannerod and G. Revy, "FLIP 1.0: a fast floating-point library for integer processors," http://flip.gforge.inria.fr/, February 2009.

[5] J. Hauser, "The SoftFloat and TestFloat Packages," available at http://www.jhauser.us/arithmetic/.

[6] G. Revy, "Implementation of binary floating-point arithmetic on embedded integer processors: polynomial evaluation-based algorithms and certified code generation," Ph.D. dissertation, Université de Lyon - ÉNS de Lyon, France, Dec. 2009.

[7] C. Bertin, C.-P. Jeannerod, J. Jourdan-Lu, H. Knochel, C. Monat, C. Mouilleron, J.-M. Muller, and G. Revy, "Techniques and tools for implementing IEEE 754 floating-point arithmetic on VLIW integer processors," in *Proceedings of PASCO'10*. New York, NY, USA: ACM, 2010, pp. 1–9.

[8] H. Saleh and J. Earl E. Swartzlander, "A floating-point fused add-subtract unit," in *51st Midwest Symposium on Circuits and Systems (MWSCAS 2008)*, 2008, pp. 519–522.

[9] H. H. Saleh and J. Earl E. Swartzlander, "A floating-point fused dot-product unit," in *2008 IEEE International Conference on Computer Design (ICCD 2008)*, Lake Tahoe, Canada, 2008, pp. 427–431.

[10] H. H. Saleh, "Fused floating-point arithmetic for DSP," Ph.D. dissertation, The University of Texas at Austin, May 2009.

[11] E. E. Swartzlander and H. H. Saleh, "FFT implementation with fused floating-point operations," *IEEE Trans. on Computers*, vol. 61, no. 2, pp. 284–288, 2012.

[12] Á. Vázquez and J. D. Bruguera, "Composite iterative algorithm and architecture for q-th root calculation," in *Proceedings of the 20th IEEE Symposium on Computer Arithmetic (ARITH-20)*, Tübingen, Germany, Jul. 2011, pp. 52–61.

[13] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[14] M. Cornea, J. Harrison, and P. T. P. Tang, *Scientific Computing on Itanium®-based Systems*. Intel Press, 2002.

[15] P. Markstein, "Accelerating sine and cosine evaluation with compiler assistance," in *Proc. of the 16th IEEE Symposium on Computer Arithmetic (ARITH)*. IEEE Computer Society, 2003, pp. 137–140.

[16] C.-P. Jeannerod and G. Revy, "Optimizing correctly-rounded reciprocal square roots for embedded VLIW cores." in *Proceedings of the 43rd Asilomar Conference on Signals, Systems, and Computers (Asilomar'09)*, Pacific Grove, CA, USA, November 2009.

[17] W. Kahan, "Lecture notes on the status of IEEE Standard 754 for binary floating-point arithmetic," May 1996, manuscript.

[18] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: SIAM, 2002.

[19] C.-P. Jeannerod and J. Jourdan-Lu, "Simultaneous floating-point sine and cosine for VLIW integer processors," February 2012, manuscript, http://hal.inria.fr/hal-00672327.

[20] C. G. Lee, "UTDSP Benchmark Suite," http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.